Notes about the various ways the !DynamicEUROPA team uses and extends EUROPA for their applications

TO DO

Here are the high level pieces that would make their models, code, and lives simpler:

- Concept of preferred values. They use this for grounding a schedule for passive resource checking, but also for their min perturbation heuristic and probably for other things.
    - ♦ Possibility: Mapping in ConstraintEngine between !PSEntityKey and PSVarValue (or an entire domain). When a user calls !ConstrainedVariable::updatePreference, the mapping in the ConstraintEngine is added to.
- Eliminate need to separate active/passive checking by using timelines AND resources:
    1. Need to be able to turn off active enforcement while keeping passive checking (currently use separate guards on separate active/passive subgoals to do this).
        - ◊ Can we use ability to allow violations so they can do this?
    2. Can decision-making and backtracking be fast (comparable to using timelines)
        - ◊ At least do this for Unary resource that behaves like a timeline, so they can use it instead. Think hard about whether they could really use this!
- They would like dynamism with ability to assign/reassign resources; switch something from one person to another, or add supervisor, remove supervisor, etc
- Add min perturbations heuristic

Resources

They have unit-capacity reusables (called claims), multi-capacity reusables (called reservations) and state resources. They use old-style (non-SAVH) resources for passive checking. They use timelines to do active checking. A multi-capacity resource is faked with a set of timelines!

For binary state resources, they use a single resource (I think this is extended to use multiple resources for a multi-state resource):

- Assume that there will never be more than 1000 things requiring a given state at any time
- Resource has bounds [0, 1000]
- Something that turns off state uses 1000 units
- Something requiring state uses 1 unit (so nothing requiring it can be scheduled when it's turned off)
- They also need special math (in their SaturatedResource? class), so that if the state gets turned off twice, it stays at 0, etc

Concerns about SAVH resources, and their solutions:

- Flow profile is too slow
    - ♦ SOLUTION: Use Timetable profile, which is fast, and will report all flaws they want because they specify all temporal variables (ie create singelton domains) before considering the profile. Note that because they are grounding, they won't actually see a benefit from the GroundedReusableProfile, which I had originally thought they needed.
- Need to not have violations reported (they use old-style trick that is equivalent, I suspect, to our !OpenWorldFVDetector)

    - SOLUTION: Use the !OpenWorldFVDetector to get no flaws reported, assuming they have infinite limits on cumulative and instantaneous production/connsumption. Even better, implement a FV variant that just doesn't report flaws at all!
- Because they use resources only for passive reporting, they don't want solver decision-resolution procedures for resource flaws (wasn't available in old-style resources)
    - SOLUTION: Use Flaw Filters that can be configured in PlannerConfig.xml.
- Need equivalent to their 'SaturatedResource?' class for state resources
    - SOLUTION: Port it to be a subclass of the new SAVH Resource, and register their SaturatedResource class as part of their own module that gets registered before the engine is started.

Min Perturbations Heuristic

Described here to best of my understanding. See also their MAPGEN paper from ICAPS 05 (attached).

Their approach is three-pronged:

1. **NDDL Changes:** each predicate has a 'reftime' temporal variable that is used to store a reference schedule that can then be accessed by disparate parts of the code.
2. **Code wrapped around core EUROPA**: fixViolations, which
    1. Sets all reference times based on current start (or a subset of activities, if desired)
    2. (If a subset of activities desired, make sure the correct subset has the 'solved' variable set to true)
    3. Run the built in solver (with custom changes mentioned below)
    4. Get list of successfully scheduled tokens based on the 'solved' variable and whether they are reallySolved()
    5. Handle issues like failure to solve, etc
    6. Fill in return-values for scheduled and unscheduled activities
3. **Solver changes:** They use the built-in solver but:
    1. *Threats* handled with custom PriorityFlawHandler (registered as PriorityMinPerturb) which uses MinPerturb decision point to resolve threats. This decision point compares potential added constraints (precedences) and orders based on how far involved tokens would have to move from reference schedule.
    2. *Open conditions* handled with NearestMerge decision point (only for some of their applications, it appears - perhaps just BedRest?), which uses reference times to determine order of merges to suggest (for each choice, it looks at how far the token would have to move from the reference time, and prefers small moves).
    3. *Unbound (temporal) variables* are left unbound (ie filtered by SolverConfig.xml), so that the above code (fixViolations etc) can bound them according to the reference schedule.
4. Temporal propagator specialization (getMinPerturbTimes) ??

Note that while this is really only min perturbations for activity start times, it does leak over into making decisions about handling threats and open conditions.

Other things they've done are intertwined with the above:

- Augment nddl predicates with 'scheduled' boolean and wrap all predicate constraints and subgoals within 'if(scheduled==true)' guard to be able to solve over-subscribed problems (ie activities can be left unscheduled to get a feasible plan)
- A 'solve' boolean is similarly used; not sure what for.